UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LUCAS DA CUNHA BUENO

IMPLEMENTAÇÃO DO ALGORITMO DO CAIXEIRO VIAJANTE COM GPGPU

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO
2022

LUCAS DA CUNHA BUENO

IMPLEMENTAÇÃO DO ALGORITMO DO CAIXEIRO VIAJANTE COM GPGPU

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de Bacharel.

Orientador: Dr. Antônio Carlos Fernandes da Silva

CORNÉLIO PROCÓPIO
2022

Dedico este trabalho a meu avô Rasberge por me ensinar o amor pela engenharia e minha avó Ana por todo apoio durante esta jornada.

AGRADECIMENTOS

Gostaria de agradecer e dedicar este trabalho às seguintes pessoas:

Minha mãe Floriana, por construir sozinha uma família maravilhosa e por sempre me ensinar a importância da educação, meu pai Carlos, por me ensinar a sempre buscar o por que das coisas e como tudo funciona, minha avó Ana por me apoiar em cada momento difícil e me ensinar resiliência, meu avô Rasberge, que permitiu que tivesse condições de realizar esta jornada, meu irmão Hugo, por me cobrar tanto e garantir que nada fosse me impedir de concluir este trajeto e minha irmã Mariana, por ser do contra e me mostrar outras formas de olhar para a vida de forma doce.

Todos os professores que passaram por minha e que foram essenciais para a formação de quem eu sou hoje, em especial ao professor Antônio que me acompanhou durante a graduação de forma exemplar, me apoiando, me orientando e me guiando para chegar até aqui e ao professor Paschoal, por me dar a chance de participar do laboratório de jogos ainda no primeiro semestre de faculdade e acreditar sempre no meu potencial.

Meu primeiro professor de física, Emerson, por ser tão didático e me ensinar a gostar da matéria e minha professora de matemática, Bel, pelos puxões de orelha que me colocaram na linha e me deram disciplina para decifrar os enigmas da vida.

Minha namorada Michele, por me apoiar e não me deixar desistir nas inúmeras vezes em que pensei que fosse fraco para enfrentar esta batalha e que me possibilitou triunfar sobre os obstáculos.

"Eu não falhei, encontrei 10 mil soluções que não deram certo.". (EDISON, Thomas).

RESUMO

BUENO, Lucas. IMPLEMENTAÇÃO DO ALGORITMO DO CAIXEIRO VIAJANTE COM GPGPU. 2022. 29 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2022.

O Problema do Caixeiro Viajante (*Traveler Salesman Problem*) é um problema de otimização com ampla margem para estudo e implementação de soluções. Com o avanço da pesquisa de aceleração de hardware e a utilização de placas de video para uso geral (GPGPU) foi possível reduzir o tempo de execução de algoritmos que demandam extensos processamentos paralelos. Neste trabalho foi implementada uma solução para acelerar um algoritmo do caixeiro viajante utilizando GPGPU em comparação com a utilização de CPUs.

Palavras-chave: Caixeiro Viajante, Processamento Paralelo, GPU, OpenCL, Aceleração de Hardware.

ABSTRACT

BUENO, Lucas. IMPLEMENTATION OF THE TRAVELING SALESMAN ALGORITHM WITH GPGPU. 2022. 29 f. Trabalho de Conclusão de Curso – Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2022.

The Traveling Salesman Problem (TSP) is an optimization problem with ample scope for studying and implementing solutions. With the advancement of hardware acceleration research and the use of graphics cards for general propose (GPGPU) it was possible to reduce the execution time of algorithms that require extensive parallel processing. In this paper, a solution was implemented to accelerate a traveling salesman algorithm using GPGPU compared to the use of CPUs.

Keywords: Traveling Salesman Problem, Parallel Processing, GPU, OpenCL, Hardware Acceletation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AG	Algoritmo Genético
CPU	*Central Processing Unit*
GPGPU	*General Propose Graphics Processing Unit*
JOCL	*Java Open Computing Language*
TSP	*Traveling Salesman Problem*
GB	*Giga Byte*
RAM	*Random Access Memory*
MHz	*Mega Hertz*
GHz	*Giga Hertz*
FPGA	*Field Programmable Gate Array*
CUDA	*Compute Unified Device Architecture*

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Serviços de entregas de diferentes naturezas como os que são empregados em restaurantes, lanchonetes, bares, farmácias, mercados e sistemas de gestão de rotas que são utilizados para ajudar no gerenciamento de grande demanda de correspondências e transportadoras, são parte importante da infraestrutura que serve linhas de produção em indústrias e atendimento aos consumidores dos comércios, promovendo o crescimento e desenvolvimento de várias camadas que se beneficiam do bom funcionamento dessa engrenagem, aproximando fronteiras com a criação de condições mais favoráveis reduzindo custos de deslocamento, consumo menor de combustíveis, menos emissão de gases poluentes na atmosfera, desgaste menor dos veículos em uso pelas empresas, menos tempo que o colaborador passa em seu itinerário podendo ser bem mais assertivo.

Rotas otimizadas significam também a possibilidade de respostas mais rápidas para veículos de emergência como Corpo de Bombeiros e Ambulâncias, em que o tempo entre o acionamento e a chegada ao endereço onde foi solicitado pode alterar completamente o resultado do chamado, e evitar vias congestionadas agiliza a chegada do socorro de maneira segura até o destino.

Além disso, concessionárias de distribuição de energia elétrica têm aplicado sistemas de gestão de rotas para posicionamento de disjuntores e chaves para alteração do roteamento da rede de transmissão, a fim de minimizar tempo que seus clientes ficam sem fornecimento em caso de emergências, como quebras na linha de transmissão ou danos em transformadores por desgastes ou descargas atmosféricas, e pela suspensão do serviço devido a manutenções na rede. De acordo com Sousa et al. (2017) em um estudo aplicado no Grupo ENERGISA, um dos principais grupos do setor elétrico brasileiro, presente em 788 cidades e atendendo por volta de 6,5 milhões de consumidores (dados de 2017), no qual foram utilizados o problema do caixeiro viajante e algoritmo genético para posicionar suas chaves e disjuntores pela rede de transmissão possibilitando a alteração da rota de transmissão de energia, melhorando os valores dos indicadores de qualidade e gerando, entre outros benefícios, retorno econômico para as distribuidoras do grupo.

De forma correlativa, Sanches (2013) defende a utilização de algoritmos evolutivos para reconfiguração de redes em sistemas de distribuição de energia elétrica, em que possibilitando a reconfiguração das redes por meio de trocas dos estados de chaves seccionadoras proporciona grande quantidade de reconfigurações possíveis para a rede. Desta maneira, em situações de interrupções no fornecimento de energia, é possível reconfigurar a rede trocando a carga entre os alimentadores. Tal reconfiguração é feita com base em um plano de reestabelecimento que consiste em "determinar um conjunto de manobras de chaves para restringir as interrupções à menor parte possível do sistema"(SANCHES, 2013).

Tendo em vista as inúmeras empregabilidades do TSP em serviços de entrega e roteamento, neste trabalho foram aplicadas técnicas de aceleração de hardware buscando reduzir o tempo de processamento do algoritmo do Problema do Caixeiro Viajante.

1.1 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado em cinco capítulos. O Capítulo 1 introduz o tema, sua abrangência e aplicações. O capítulo 2 apresenta o referencial teórico sobre o TSP, como as formulações do problema, a construção das rotas, os métodos de recombinação, o algoritmo utilizado, a heurística empregada, a otimização e paralelização da resolução e o perfilamento do algoritmo. O capítulo 3 trata do desenvolvimento do trabalho, com a construção do algoritmo base, o perfilamento buscando encontrar os pontos críticos e a remodelação para que o algoritmo seja executado de forma paralela. O capítulo 4 compreende os resultados obtidos, a análise dos dados coletados, bem como uma discussão sobre as razões pelas quais estes resultados foram alcançados. O capítulo 5 apresenta a conclusão deste trabalho, sugestões de trabalhos futuros que possam utilizar este como base para estender o assunto e explorar diferentes tecnologias e as considerações finais.

2 REFERENCIAL TEÓRICO

Este capítulo discorre sobre o referencial teórico do tema abordado no trabalho a fim de fundamentar as informações e os dados apresentados.

2.1 O PROBLEMA DO CAIXEIRO VIAJANTE

O Problema do Caixeiro Viajante, do inglês *Traveling Salesman Problem* (TSP), consiste na situação de um caixeiro viajante que deseja visitar apenas uma vez cada uma das cidades de uma lista de n cidades e retornar ao ponto de partida ao final, onde o custo para sair de uma cidade A e ir para uma cidade B é somado a cada ponto e o somatório final é o custo para percorrer a sequência de cidades escolhida (HOFFMAN et al., 2013).

Esta descrição se refere a um problema típico de otimização combinatória de complexidade tipo NP, problemas "que não podem ser resolvidos por nenhum algoritmo conhecido utilizando métodos determinísticos, mas quando uma solução é proposta é simples de verificar a veracidade"(ABDULRAZAQ et al., 2019), ou seja, para a resolução de um problema com n cidades, existem possíveis $(n-1)!/2$ rotas e que até o presente momento não foi encontrada uma solução que seja efetiva e que não seja a busca exaustiva, em que cada possível caminho é analisado e comparado na busca pelo menor. Quando n cresce, o tempo necessário para analisar todas as possibilidades se torna inviável até para os computadores mais rápidos da atualidade, e por não possuir qualquer solução em tempo polinomial o torna um problema não determinístico do tipo NP-Completo (DANG; ZHANG, 2005).

Assim como na pesquisa de Syambas, Salsabila e Suranegara (2017), buscou-se desenvolver algoritmos heurísticos que fossem capazes de encontrar resultados que se aproximam suficientemente de um resultado em tempo polinomial. O emprego de algoritmo genéticos, que são algoritmos modelados com base em processos biológicos em que gerações anteriores passam características para as gerações posteriores de forma a realizar uma seleção artificial, é o principal método de análise deste tipo de problemática (FRAHADNIA, 2009). Avançando em algoritmos mais eficientes, o tempo computacional,isto significa, o quão extenso é o processo de computar tais dados devido a sua complexidade e amplitude, se torna o maior limitante para aplicação destas soluções em situações em que se faz necessário respostas mais rápidas e conjuntos de dados extremamente massivos. Assim, este trabalho buscou aplicar técnicas de processamento paralelo para reduzir o tempo de processamento do problema, designado como tempo de execução do algoritmo.

2.2 FORMULAÇÕES DO PROBLEMA DO CAIXEIRO VIAJANTE

O Problema do Caixeiro Viajante possui diferentes formulações que foram concebidas visando diversas aplicações e otimizações baseadas no ambiente onde seriam aplicadas (SUN et al., 2011). Neste trabalho a formulação encontrada no algoritmo de Lin-Kernighan foi abordada por ser uma das interpretações mais eficientes para resolução do TSP (LIN; KERNIGHAN, 1973) e que mais se aproxima dos propósitos desta pesquisa, em razão de sua eficiência e possibilidade de paralelização.

A pesquisa de Fischer e Merz (2005) define o algoritmo de Lin-Kernighan como uma das melhores heurísticas para resolver o TSP e descreve o problema como "a busca por uma rota com um custo ótimo para um dado número de cidades em que cada uma destas cidades seja visitada apenas uma única vez", e mais recentemente, Caiano (2018) afirma que "um dos

algoritmos melhor sucedido para o famoso problema do caixeiro viajante é o algoritmo de Lin-Kernighan, sendo considerado um dos métodos mais eficazes para obter soluções ótimas ou quase ótimas."

O modelo matemático empregado pode ser escrito na forma da Equação 1 representada abaixo:

$$C = \sum_{i=1}^{n} d_{ij} \tag{1}$$

Sendo:

C - O custo de percorrer todos as cidades da rota
n - Número de cidades a serem visitadas
i - Ponto de início ao longo da rota variando de 1 a n
j - Ponto de chegada ao longo da rota variando de 1 a n
d_{ij} - Distância da cidade i até a cidade j

Em que cada cidade só pode ser visitada uma única vez e nenhuma cidade pode não ser visitada.

2.2.1 CONSTRUÇÃO DA ROTA COM BUSCA LOCAL

Como descrito por Davendra (2010), a construção de cada rota finaliza quando uma solução é encontrada e não é possível incrementá-la. Algoritmos para construção destas rotas utilizam diferentes abordagens e as mais comuns estão descritas abaixo.

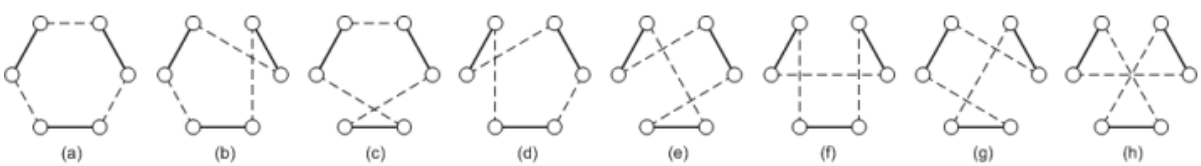
2.2.1.1 2-OPT E 3-OPT

O método 2-opt consiste em "remover duas conexões entre cidades da rota reconectá-las entre si, repetindo esta operação enquanto o resultado da mesma for uma rota final mais curta"(BLAZINSKAS; MISEVICIUS, 2011). A remoção e reconexão da rota até não obter mais ganho sucede em uma rota denominada mínimo local em tempo polinomial de complexidade $O(N^2)$.

O mecanismo da abordagem 3-opt atua de forma similar, mas ao invés de remover duas conexões, três conexões são quebradas e re-roteadas. Este procedimento provê soluções melhores, mas é consideravelmente mais lenta, com complexidade $O(N^3)$.

Ilustrando o artifício citado acima em um cenário hipotético com 6 cidades, a técnica deriva oito possibilidades de reconexão, conforme esclarecido na Figura 1, o algoritmo itera o mecanismo enquanto as recombinações resultam em pelo menos uma rota cujo comprimento seja reduzido em relação ao original.

Figura 1 – Possíveis recombinações da operação de 3-opt



Fonte: Blazinskas e Misevicius (2011)

2.2.1.2 K-OPT

A técnica do k-opt, com complexidade $O(n^k)$, consiste na expansão dos métodos abordados acima, mas por sua vez são realizadas k trocas até que alguma não resulte em aprimoramento do resultado (HELSGAUN, 2009).

2.3 ALGORITMO GENÉTICO PARA O TSP

Esta seção apresenta a aplicação de algoritmo genético (AG) para resolução do TSP.

AG é um método de otimização que utiliza uma abordagem estocástica para procurar aleatoriamente boas soluções para um problema específico, que se baseia no princípio Darwiniano de que o individuo que melhor se adapta torna-se mais forte e isto implica em indivíduos com maior probabilidade de sobreviver e transmitir suas boas características genéticas para a próxima geração. Os indivíduos são chamados neste cenários de cromossomos, partes de uma população que constituem uma possível resposta para o problema que no âmbito do TSP, compõem rotas candidatas a solucionar o problema e o *fitness*, parâmetro referente a uma pontuação dada a fim de diferenciar os cromossomos mais propensos a se reproduzir, *fitness* maiores, dos que serão descartados, *fitness* menores (RAZALI; GERAGHTY et al., 2011).

Uma sequência de passos básica de operação do AG é a seguinte:

1. Inicialização: Gera um população aleatória de N cromossomos.
2. *Fitness*: Calcula o *fitness* de todos os cromossomos
3. Cria uma nova população:
   a) Seleção: Utiliza o método de seleção para separar dois cromossomos da população.
   b) *Crossover*: Realiza o cruzamento dos cromossomos selecionados.
   c) Mutação: Realiza o processo de mutação no resultado obtido.
4. Substituição: Substitui a antiga população pela gerada.
5. Teste: Se alguma das condições de parada for satisfeita. Se sim, o processo é finalizado, senão, retorna a melhor solução na população de soluções atuais para o Passo 2.

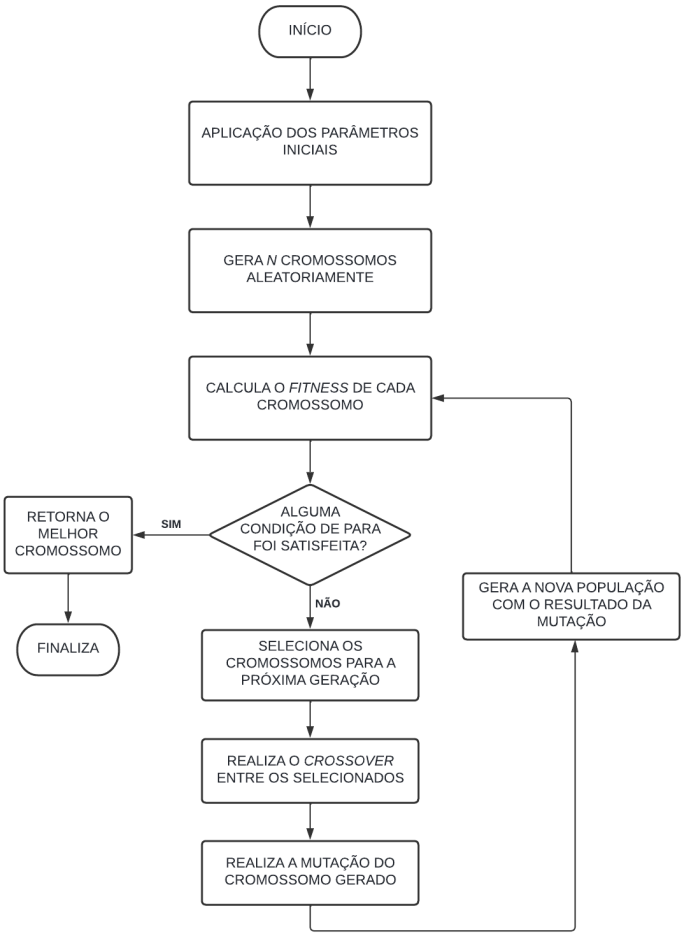
O processo acima enumerado pode ser melhor esclarecido pelo diagrama apresentado na Figura 2.

O processo do AG necessita de parâmetros iniciais que determinam características quantitativas do método, abaixo está exposto tais parâmetros e suas particularidades, baseados nos critérios estabelecidos em Rexhepi, Maxhuni e Dika (2013).

- Número máximo de gerações: Cada iteração da sequência de passos acima é conhecida como geração e o número máximo de gerações é uma das condições de parada do algoritmo.
- População: Conjunto de cromossomos que representam o espaço de possíveis candidatos a solução.
- Taxa de *Crossover*: Probabilidade de o processo de *crossover* ser executado, procedimento este que combina partes de um cromossomo com outro a fim de gerar um descendente para a próxima geração que herda traços de ambos genitores.
- Taxa de Mutação: Probabilidade de que após o *crossover* seja realizada uma mutação no cromossomo, ou seja, trocas arbitrárias de posições entre partes de um cromossomo com finalidade de evitar que todos os membros de uma população caiam em um mínimo local.

Tais propriedades não são um consenso e variam de acordo com a aplicação do AG, tamanho da população, nível de precisão do resultado esperado, entre outros. Em Saenphon (2018), por exemplo, em que o foco foi a comparação entre diferentes algoritmos utilizando parâmetros constantes entre eles, foi utilizado um total de 500 gerações, taxa de *crossover* de 12%, taxa de mutação de 8% e uma população de 100 cromossomos. Ao passo que em

Figura 2 – Diagrama de execução do AG



Fonte: Adaptado de Odili, Kahar e Ahmad (2016)

Singh et al. (2018), em que o foco foi propor um novo método de realização do *crossover*, foi empregado o total de 50 gerações, taxa de *crossover* de 85%, taxa de mutação de 5% e uma população de 100 cromossomos.

2.4 CONJUNTOS DE COORDENADAS A SEREM PERCORRIDAS PELO TSP

Para construção dos trajetos a serem percorridos pelo algoritmo no intuito de calcular a distância entre os pontos de parada, foram utilizados conjuntos de coordenadas, também chamados de *Dataset's*, disponibilizados pela Universidade Heidelberg na Alemanha, chamado TSPLIB (REINELT, 1995). Neste trabalho, considera-se tais pontos dispostos em um plano, em que as distâncias Euclidianas serão calculadas entre eles. O cálculo da distância Euclidiana entre dois pontos no plano é dado pela Equação (2) mostrada abaixo:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

Sendo:
D - A distância entre os pontos.
x_1 e y_1 - As coordenadas do ponto de partida.

x_2 e y_2 - As coordenadas do ponto de chegada.

Além disso, foi adotado a condição do problema ser simétrico, ou seja, a distância para transitar entre um dado ponto A e um segundo ponto B é a mesma se o trajeto inverso, de B até A, for percorrido.

A Tabela 1 apresenta os *Dataset's* presentes neste trabalho e a principal intenção em utilizar estes conjuntos de dados é ter uma base sólida de pesquisa em razão dos mesmos serem compartilhados e continuamente alimentados pelos autores desde 1995, e, portanto, disponível para que os dados aqui apresentados possam ser confrontados e futuramente utilizados como comparação.

Tabela 1 – *Dataset's* utilizados.

Nome do *Dataset*	Número de cidades
ATT48	48
RAT575	575
NRW1379	1379
BRD14051	14051
D18512	18512

Fonte: Reinelt (1995)

2.5 A HEURÍSTICA DE LIN-KERNIGHAN

A fim de entender a metodologia utilizada para concepção do algoritmo, nesta seção será descrita a heurística utilizada e o algoritmo empregado.

A heurística de Lin-Kernighan utiliza um conjunto de soluções iniciais em que serão aplicadas um primeiro aprimoramento utilizando o método 2-opt, iniciando então a sequência de manipulações a cada iteração até que alguma das condições de término, como ser atingido o número máximo de gerações, seja satisfeita (CRIŞAN; NECHITA; SIMIAN, 2021). O conceito citado pode ser melhor compreendido analisando o Algoritmo 1.

Algoritmo 1: TSP com Algoritmo Genético

Entrada: Conjunto de cidades e as respectivas distâncias entre elas R,
o número máximo de gerações N,
o tamanho da população de cromossomos P,
a taxa de *crossover* C
e a taxa de mutação M.
Saída: Cromossomo que representa um ótimo local
início
  Solucione P cromossomos iniciais para a população inicial;
  repita
    Calcule o *fitness* dos cromossomos atuais;
    Selecione os melhores cromossomos;
    se *Condição de realização do crossover* C *for atingida* **então**
      Realize o *crossover* entre os pares de cromossomos selecionados;
    fim
    se *Condição de realização da mutação* M *for atingida* **então**
      Realize a mutação do cromossomo obtido;
    fim
    Calcula o *fitness* do cromossomo resultante;
    se *fitness calculado é melhor que o atual ótimo local* **então**
      Elenca o cromossomo como ótimo local;
    fim
    ;
  até *Geração Atual* $= N$;
fim

2.6 OTIMIZAÇÃO DO ALGORITMO

Apesar de ser concebido para otimizar a busca de soluções, ainda existem pontos a serem estudados buscando aumentar ainda mais a eficácia do método.

Detalhando o algoritmo descrito, evidenciou-se a possibilidade de ganhos de desempenho utilizando a paralelização, em razão de que para cada geração é calculada a distância entre cada uma das cidades da rota. Deste modo buscou-se artifícios para obter tais ganhos.

2.6.1 PARALELIZAÇÃO

Seguindo o paradigma descrito no Algoritmo 1, no qual a estrutura é executada de forma sequencial, foi proposto neste trabalho a paralelização, isto é, "tirar vantagem de processadores com múltiplos núcleos e balancear de forma eficiente o conflito entre o aumento no tamanho do problema e a eficiência computacional na busca por soluções satisfatórias"(WEI et al., 2021). Segundo os autores citados, a arquitetura de computadores modernos, baseados em máquinas de Turing, são seriais por natureza, ou seja, executam as tarefas de forma sequencial e mesmo com o advento de novas tecnologias e processadores com múltiplos núcleos, ainda têm dificuldade em obter ganhos expressivos de desempenho em algoritmos que não foram projetados para obter vantagem destes núcleos a mais.

O diagrama ilustrado na Figura 3 exemplifica a sucessão de operações para o cálculo das distancias entre todas as N cidades de uma rota de forma sequencial, realizando apenas uma por vez. Se cada operação demorar um tempo t para ser executada, será necessário um

tempo Nt para cada geração, enquanto o modelo apresentado na Figura 4 idealiza a realização dos cálculos de forma paralela, no qual apenas uma operação será realiza, ou seja, levaria um tempo t para ser executado. Este ganho hipotético com a paralelização é irreal na maioria das vezes pois fica restrito a limitações impostas pelo *hardware* que está processando estes dados e que conta com um número finito de núcleos.

Figura 3 – Diagrama de execução de cálculos de forma sequencial

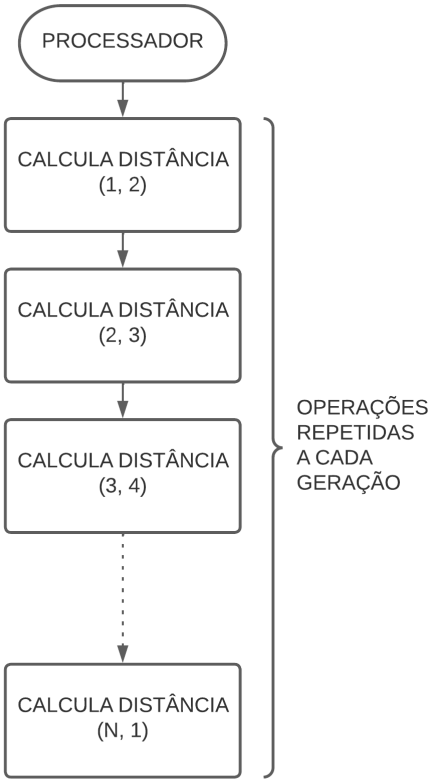
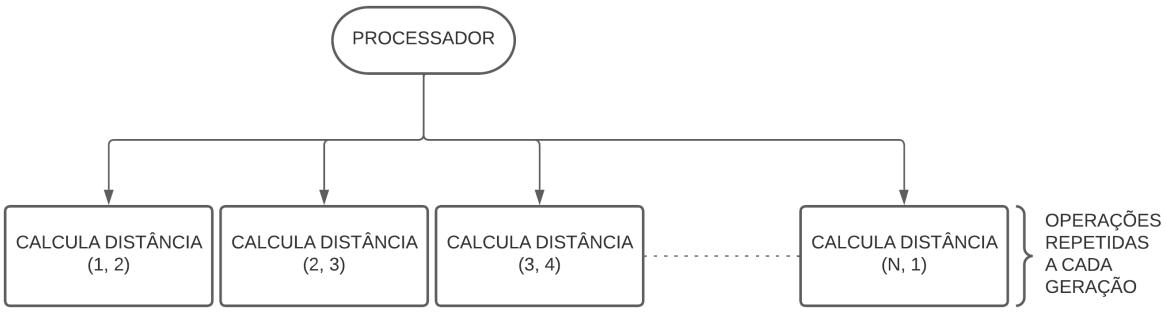


Figura 4 – Diagrama de execução de cálculos de forma paralela



2.6.2 *LOOP UNROLLING*

Em razão das limitações impostas pelo *hardware*, exigiu-se a utilização da técnica de *Loop Unrolling*, que de acordo com Oo e Chaikan (2021), "consiste em uma técnica de tentar

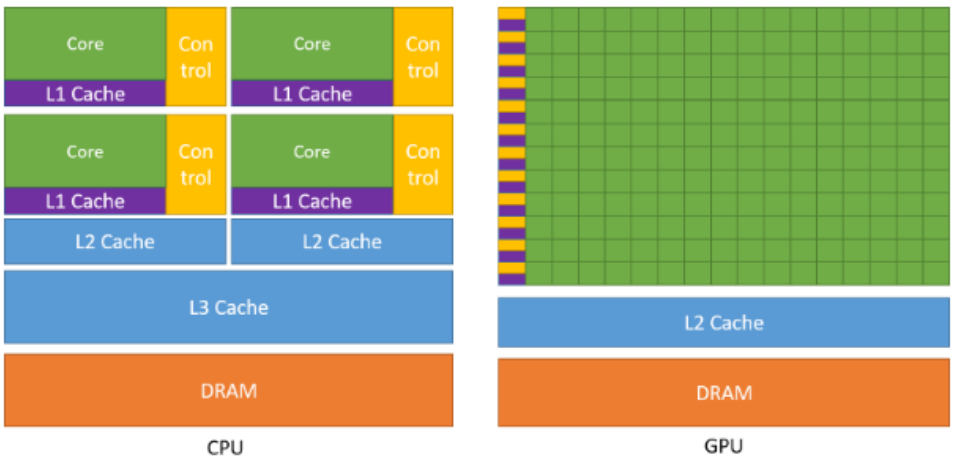
minimizar a sobrecarga e reduzir o espaço de iteração das repetições". Tal artifício permite dividir uma sequência de repetições em k séries menores, sendo $k > 0$ e $k <= N$, sendo N o número de cidades da rota, em que k é conhecido como *Unroll Factor* e seu valor é variável de acordo com o equipamento utilizado e o conjunto de dados a ser processado. Obter um valor de k ideal é o mesmo que encontrar um valor que, dividindo o número total de operações a serem realizadas pelo algoritmo a cada geração por ele, tem-se por resultado k sequências de operações das quais o processador utilizado entrega a maior eficácia na resolução dos cálculos de forma paralela.

2.6.3 UTILIZANDO GPU'S AO INVÉS DE CPU'S

Utilizadas nos mais variados dispositivos eletrônicos como computadores, celulares e *tablets*, as Unidades de Processamento Central (CPU), são o coração do processamento de dados e são responsáveis pela computação dos dados em tais aparelhos, de forma análoga, Bräunl (2008) descreve a CPU "como um relógio mecânico. Um grande número de componentes que interagem entre si, seguindo o ritmo de um oscilador central, em que cada parte tem que se mover exatamente no momento correto". CPU's modernas são equipadas com 4, 8, 16 ou até mais núcleos executam tarefas de forma simultânea a fim de aumentar o desempenho do equipamento. Tamanha vantagem tornou CPU's com múltiplos núcleos "... têm se tornado produtos inevitáveis do desenvolvimento tecnológico"(WEI et al., 2021).

Apesar do potencial ganho de desempenho empregando a técnica de *Loop Unrolling* utilizando a CPU, foi proposto o estudo de possíveis ganhos ainda mais expressivos utilizando uma Unidade de Processamento Gráfico (GPU), outro dispositivo presente em grande parte dos computadores atuais e que dispõe de um número significativamente maior de núcleos. A GPU, conforme descrita por Syberfeldt e Ekblom (2017), é "um processador especializado originalmente projetado para remover a carga de renderização de gráficos 3D da CPU"e complementam que a arquitetura destas unidades "permitem paralelizações massivas através de milhares de núcleos de processamento". A utilização do poder de processamento paralelo destes dispositivos para processamento de propósito geral é conhecida pelo termo GPGPU (sigla em inglês *General-Purpose computation on GPUs*, Computação de Propósito Geral em Unidades de Processamento Gráfico) (TSUDA, 2011).

Figura 5 – Ilustração comparativa da arquitetura da CPU e da GPU



Fonte: NVidia (2021)

O algoritmo primeiramente executado de forma sequencial no processador AMD Ryzen 5950X, que possui 16 núcleos operando no máximo a 4.90 GHz (AMD, 2022), teve suas partes com potencial de paralelização executadas na GPU NVidia RTX 3090 com 10496 núcleos que atingem picos de até 1,70 GHz(NVIDIA, 2022), que embora individualmente não tenham o mesmo poder de processamento dos núcleos do processador da AMD, se utiliza da massa de núcleos 655 maior e de uma arquitetura voltada para este tipo de processamento para suprimir esta desvantagem nestes cenários de paralelização, que de acordo com Sethumadhavan, Narayanasamy e Gopalakrishnan (2016), para este tipo de aplicação, mesmo que o processador contasse com um número maior de núcleos, a CPU não seria capaz de superar a GPU devido a sua arquitetura, e além disso, adicionar estes núcleos a mais seria custoso, uma vez que uma placa de vídeo com mais de 1000 núcleos tem valor próximo de um processador com 8 núcleos.

Diferentemente da CPU que utiliza memórias DDR4 operando a 3200 MHz que atingem picos de transmissão de 25,6 GB por segundo (CRUCIAL, 2022), a placa de vídeo utilizada neste trabalho possui uma arquitetura diferente e mais moderna, operando com memórias GDDR6X capazes de alcançar velocidades de até 1152 GB por segundo (MICRON, 2022).

2.6.4 IDENTIFICANDO PONTOS CRÍTICOS

Em virtude de que CPUs e GPUs possuem vantagens e desvantagens, fez-se necessário analisar o algoritmo em busca dos segmentos que se beneficiam de serem executados pela CPU de forma sequencial e os que são favorecidos pela paralelização massiva oferecida pela GPU.

2.6.4.1 *PROFILING*

Conhecida como *Profiling*, a técnica de perfilar o algoritmo, de acordo com Zaparanuks (2018), consiste em analisar o consumo de recursos durante a execução de suas tarefas, permitindo então distinguir trechos que demandam mais tempo para serem concluídos, com maior custo computacional e trechos que são executados de forma repetitiva por longos períodos, neste caso, com potencial para serem paralelizados. Neste trabalho foram analisados os fatores de *Self Time*, tempo que o processador leva para executar um método específico, e o *Total Time*, tempo total para execução do algoritmo.

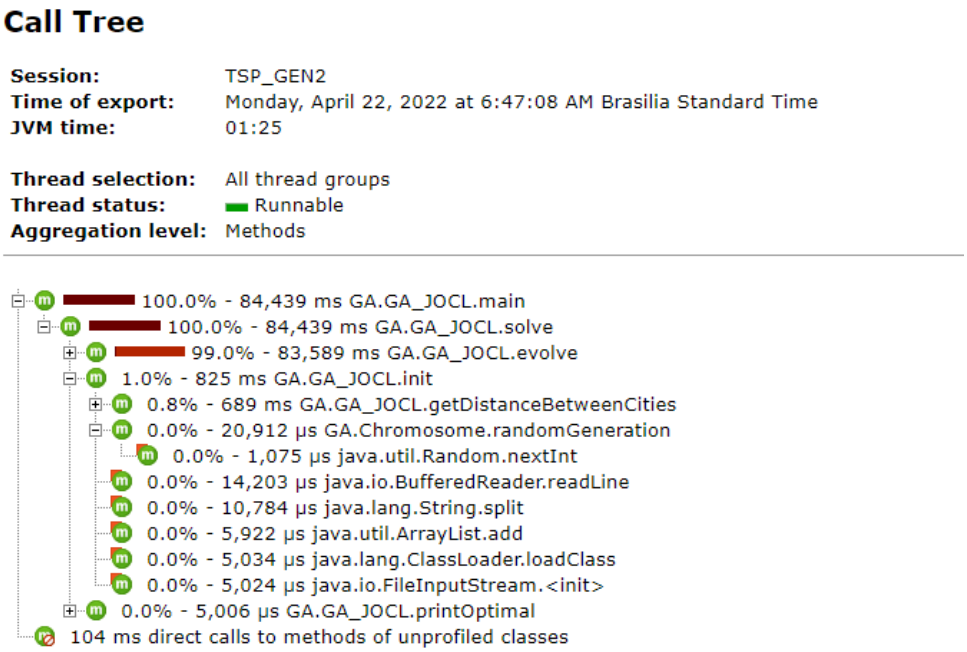
O processo de *profiling* do algoritmo foi feito utilizando uma ferramenta de perfilamento chamada JProfiler (2022), uma suíte de soluções para análise de desempenho de aplicações Java que executa o *profiling* de forma não intrusiva, com a capacidade de computar o tempo despendido pela CPU ou GPU para processar cada comando e contabilizar quantas vezes cada instrução foi acionada, indicando assim os principais trechos de código candidatos a serem paralelizados, além de possibilitar a repetição dos testes sob as mesmas condições de modo a obter uma média dos resultados. Neste trabalho, cada teste foi executado 50 vezes com objetivo de que interferências e erros aleatórios sejam absorvidos.

A Figura 6 representa uma extração de dados do JProfiler quando executado o *profiling* da resolução do *dataset* ATT48 utilizando apenas uma *thread* com finalidade de elucidar o formato de saída das informações que serão futuramente analisadas.

Cada linha da Figura 6 exibe a porcentagem do tempo de execução exigido pelo grupo de métodos, o tempo decorrido para sua execução e o nome dos métodos, em síntese, o *Total Time* é apresentado na primeira linha, neste exemplo, o tempo de 84.439 milissegundos, ou 84,439 segundos.

O *Self Time* por sua vez é exibido na linha 5, onde o método *getDistanceBetweenCities*, responsável pelo cálculo da distância entre os nós do *dataset* a fim de compor a contabilização

Figura 6 – Ilustração da extração de dados do JProfiler



Fonte: JProfiler (2022)

do *fitness* do cromossomo, é exibido. Ainda que aparente ser um valor pequeno em comparação com o *Total Time*, o tempo de 689 milissegundos do *Self Time* será considerado um ponto crítico a ser analisado no próximo capítulo.

2.7 TRABALHOS RELACIONADOS

Crişan, Nechita e Simian (2021) realizaram um estudo que analisou a influencia de elementos estruturais medindo o desempenho computacional pela qualidade do resultado na resolução do TSP com as heurísticas da colonia de formigas e a de Lin-Kernighan, buscando provar que a estrutura da instância do TSP influencia na qualidade da solução, mas sem analisar o custo computacional e o quão próximo a solução apresentada está de uma solução ótima. Neste trabalho foi analisado apenas a heurística de Lin-Kernighan para a resolução do TSP, visto que o foco é a avaliação do custo computacional e não a efetividade do método em comparação com os demais.

Oo e Chaikan (2021) expuseram o efeito da utilização da técnica de *Loop Unrolling* visando aumento da eficiência energética para resolução do algoritmo de Strassen, que utiliza de forma massiva a multiplicação de matrizes quadradas, em uma arquitetura de memória compartilhada, obtendo resultados de economia de energia devido a redução de transferência de dados e acesso a memória. Neste trabalho foi utilizada a técnica buscando diminuir o tempo entre as execuções e extrair o máximo de desempenho da GPU, analisando também o impacto de diferentes valores para o *Unroll Factor* em cada *dataset*, visando o fator ótimo que resultou no menor tempo necessário para o cálculo do *fitness* de cada cromossomo.

Singh et al. (2018) propuseram um novo operador de *crossover* e um novo método de inicialização da população para resolução do TSP com algoritmos genéticos. O operador de *crossover* proposto diminui a chance de membros redundantes na população baseado na troca de informações entre os cromossomos, em que a combinação dos cromossomos pais

tem mais probabilidade de produzir filhos melhores e que ainda não tenham sido testados. O método de inicialização da população proposto aumenta a diversidade entre as populações e remove a redundância resultada da inicialização aleatória pela utilização de teoria de grupos de forma iterativa para formar uma tabela de resultados que não necessita de procedimentos para checagem de elementos repetidos. Neste trabalho a forma de escolha da população foi de forma aleatória e o *crossover* foi feito a partir da recombinação dos pais, quebrando os cromossomos de forma aleatória de recombinando-os.

Baumann, Noack e Steinke (2021) compararam a melhoria de performance alcançada pela implementação de OpenCL em processadores Intel utilizando a versão de código aberto e uma versão proprietária desenvolvida pela Intel. Neste trabalho foi utilizado o JOCL, a versão de código aberto do OpenCL elaborada para ser implantada em algoritmos que utilizam a linguagem de programação Java, visto que o algoritmo seria executado tanto pela CPU quanto pela GPU.

Syberfeldt e Ekblom (2017) comparam a implementação de paralelização de algoritmos evolutivos utilizando CPUs e GPUs, conferindo também os resultados de múltiplas execuções independentes em paralelo, com pretensão de prover dados sobre qual plataforma é mais eficiente e mais rápida. Neste trabalho foi realizada tal comparação mas por sua vez utilizando técnicas de algoritmos genéticos para resolução do TSP e sem a necessidade de execuções múltiplas em paralelo pois buscou-se obter o máximo de uso do *hardware* disponível para cada execução.

Blazinskas e Misevicius (2011) utilizam a heurística de Lin-Kernighan aliada com os métodos *2-opt*, *3-opt* e *4-opt* com perturbações introduzidas pela técnica de *k-swap-kick*, além de combinações destes, para resolver de forma mais rápida o TSP. Neste trabalho o foco foi a utilização apenas do método *2-opt* pois a complexidade de aplicação dos demais métodos de forma combinada com paralelização não seria interessante para o escopo do trabalho.

3 DESENVOLVIMENTO

Este capítulo descreve as etapas de desenvolvimento deste trabalho, tendo em consideração a construção de um algoritmo base para determinação dos pontos críticos a serem paralelizados pelo processo de *profiling*, remodelação do algoritmo para executar tais pontos via GPGPU de forma paralela e então retomar a técnica de *profiling* para comparar os dados obtidos antes e depois das modificações aplicadas.

3.1 CONSTRUÇÃO DO ALGORITMO

Como base foi utilizado o projeto desenvolvido por Intgrp (2020), que utiliza a base de AG para resolução do TSP de forma totalmente sequencial, desenvolvido com a linguagem de programação Java. De início, foram aplicadas modificações para que fossem utilizados os *datasets* ATT48, RAT575, NRW1379, BRD14051 e D18512.

3.1.1 *PROFILING* PRÉ-PARALELIZAÇÃO

O resultado obtido pelo processo acima indicou que o maior tempo gasto pela CPU para execução do algoritmo é durante a contabilização do *fitness*. Este cálculo é realizado pela soma das distâncias Euclidianas (Equação (2)) entre os pontos do trajeto a fim de obter a distância total para percorre-lo, portanto, este processo é repetido a cada geração para cada cromossomo da população. O comparativo da porcentagem de tempo de CPU gasto e o número de vezes que a distância entre postos foi executada estão apresentados na Tabela 2. Sendo assim, fica definido este cálculo como ponto crítico a ser paralelizado.

Tabela 2 – Tempo de CPU para o cálculo da distância Euclidiana.

Dataset	% do Tempo Total de CPU	Número de Instâncias
ATT48	4,67%	249.654
RAT575	14,21%	2.990.512.228
NRW1379	23,17%	7.170.842.134
BRD14051	73,15%	73.065.629.315
D18512	96,37%	96.262.965.616

Fonte: Extraído dos resultados do JProfiler.

3.1.2 REMODELAÇÃO DO ALGORÍTIMO

O trecho determinado como crítico pelo *profiling* é executado por um laço de repetição, desta forma, abrindo então precedente para aplicação da técnica de *Loop Unrolling*, com propósito de paralelizar esta iteração em iterações menores e que por sua vez serão executadas pela GPU ao invés da CPU.

3.1.2.1 APLICAÇÃO DA TÉCNICA DE *LOOP UNROLLING* E EXECUÇÃO VIA GPU

Esta técnica de particionamento de um grande laço de repetição em k laços menores que são executadas em paralelo tem como precedente a definição do k, chamado neste contexto

de *Unroll Factor*. A determinação deste fator foi feita de forma empírica, testando valores de forma crescente com intuito de alcançar uma curva de resultados em que o fator ideal foi fixado pelo ultimo ponto em que a curva cresceu. Deste ponto em diante, fatores limitantes no equipamento como número de núcleos de processamento e tamanho da memória alocada tornam-se um obstáculo. Para realização dos testes com objetivo de encontrar o *Unroll Factor* ideal foi necessário implementar a seção crítica do algoritmo que doravante será executado via GPGPU.

Para implementação do cálculo do *fitness* na GPU foi empregada a linguagem *OpenCL* (*Open Computing Language*) que é uma plataforma de código aberto para programação paralela que permite que um programa que está sendo executado pela CPU implemente um ambiente *OpenCL* e envie comandos para serem executados em dispositivos compatíveis (BAUMANN; NOACK; STEINKE, 2021), neste caso, a GPU. A execução dos comandos *OpenCL* dentro de um programa Java é feita mediante utilização de uma biblioteca chamada JOCL, que viabiliza a comunicação entre o programa e a GPU.

Para cada *dataset* estudado foi realizada uma bateria de testes para determinação do *Unroll Factor* e os dados apresentados na Tabela 3, Tabela 4, Tabela 5, Tabela 6 e Tabela 7.

Tabela 3 – *Unroll Factor* para o *dataset* ATT48.

Unroll Factor	Tempo para cálculo do *fitness* (ms)
30	0,111201
40	0,085420
48	0,061784

Fonte: Extraído dos resultados do JProfiler.

Tabela 4 – *Unroll Factor* para o *dataset* RAT575.

Unroll Factor	Tempo para cálculo do *fitness* (ms)
500	0,112154
550	0,097548
575	0,071304

Fonte: Extraído dos resultados do JProfiler.

Tabela 5 – *Unroll Factor* para o *dataset* NRW1379.

Unroll Factor	Tempo para cálculo do *fitness* (ms)
1.100	0,109154
1.200	0,098013
1.300	0,099512

Fonte: Extraído dos resultados do JProfiler.

Tabela 6 – *Unroll Factor* para o *dataset* BRD14051.

Unroll Factor	Tempo para cálculo do *fitness* (ms)
8.000	0,104572
10.000	0,100145
12.000	0,106487

Fonte: Extraído dos resultados do JProfiler.

Tabela 7 – *Unroll Factor* para o *dataset* D18512.

Unroll Factor	Tempo para cálculo do *fitness* (ms)
14.000	0,108412
15.500	0,101046
17.000	0,106480

Fonte: Extraído dos resultados do JProfiler.

Para os *datasets* ATT48 e RAT575, foi evidenciado a queda no tempo necessário para o cálculo do *fitness* para todos os fatores utilizados, chegando ao valor máximo que é limitado pelo tamanho do *dataset*. Em contraste, no caso do NRW1379, aplicando os fatores 1100 e 1200 foi constatado um ganho de desempenho, mas aumentando este número para 1300 foi observado um aumento no tempo necessário para o cálculo, indicando ter alcançado um valor em que a divisão do laço de repetição encontrou um limitante no processamento via GPU. De forma similar, os *datasets* BRD14051 e D18512 apresentaram os melhores resultados com os fatores 10.000 e 15.500, respectivamente. Com os dados em posse, definiu-se os fatores em 48 para o ATT48, 575 para o RAT575, 1200 para o NRW1379, 10.000 para o BRD14051 e 15.500 para o D18512.

4 RESULTADOS

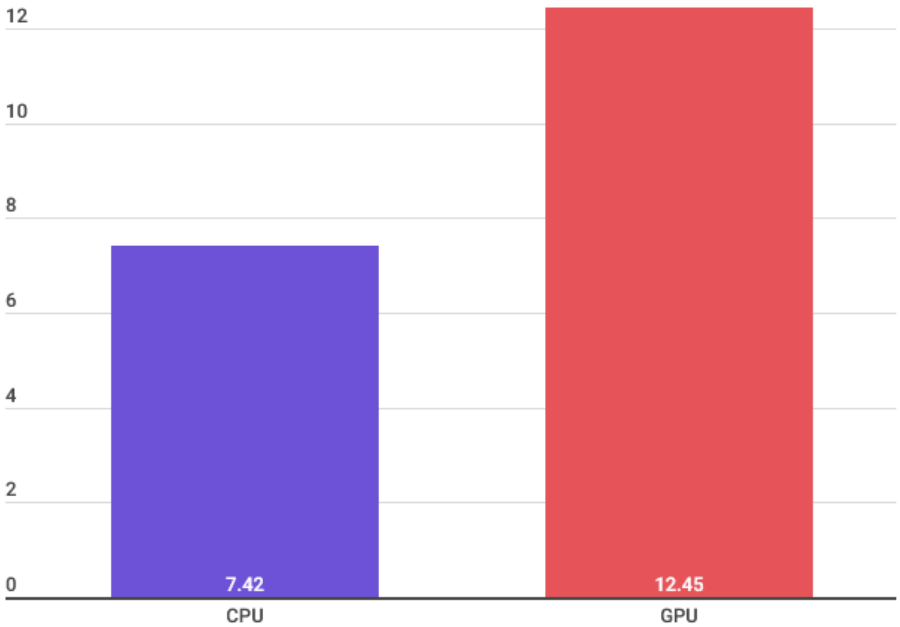
Posteriormente a aplicação da técnica de *Loop Unrolling* e de destinar o segmento paralelizado para ser executado via GPGPU, o processo de *profiling* foi novamente realizado com o objetivo de coletar os dados para observar os possíveis benefícios trazidos pelos procedimentos que o algoritmo foi submetido. Neste capítulo serão analisados os resultados alcançados e avaliar o desempenho da metodologia aplicada, para tanto, foram averiguados os ganhos na execução do processo completo do algoritmo e posteriormente os ganhos obtidos apenas no processo de cálculo da distância Euclidiana.

4.1 COMPARATIVO ENTRE CPU E GPU

A seguir serão apresentados os dados colhidos pelo *profiling* do algoritmo completo para solução do TSP obtidos durante a execução do problema na CPU e na GPU.

4.1.1 ANÁLISE DO *DATASET ATT48*

Figura 7 – Tempos para execução completa do *dataset* ATT48 (s)

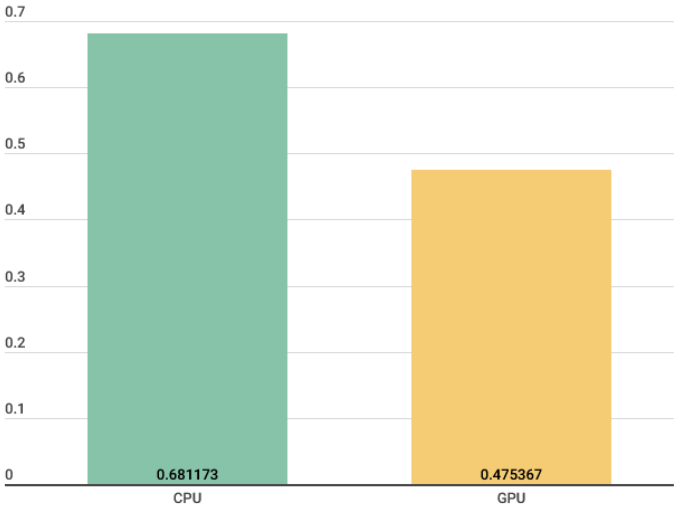


Fonte: JProfiler (2022)

Para o *dataset* ATT48 o resultado encontrado foi negativo, em que o processo completo de execução foi 67,74% mais lento quando executado pela GPU, conforme ilustrado no gráfico apresentado na Figura 7. Embora seja o menor *dataset* abordado neste trabalho, este cenário se mostrou controverso uma vez que ao analisar os dados obtidos, ele apresenta um aumento no tempo total quando executado via GPU quando comparado com o mesmo cenário sendo executado via CPU, um retrocesso que se analisado superficialmente inviabiliza a aplicação

do método. Visto que grande parte da resolução do TSP ainda é executada pela CPU em decorrência da vantagem oriunda de sua arquitetura para execuções sequenciais, foi analisado também o rendimento do procedimento aplicado apenas para o cálculo da distância Euclidiana, também chamado de *Self Time*, onde efetivamente houve execução de tarefas pela GPU. Quando analisados os dados demonstrados na gráfico apresentado na Figura 8, o mesmo *dataset*, exibe uma queda de 0,205 milissegundos por iteração ou 24,79%.

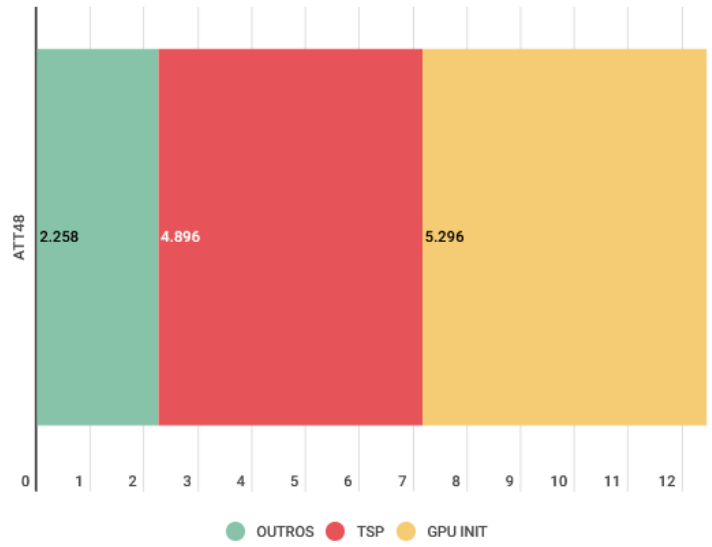
Figura 8 – Tempos para execução apenas da distância Euclidiana do *dataset* ATT48 (ms)



Fonte: JProfiler (2022)

A ferramenta de *profiling* JProfiler possibilita também extrair os dados dos tempos para cada etapa do processo executado pela GPU, desta forma, o gráfico mostrado na Figura 9 apresenta a divisão do tempo despendido para execução do algoritmo para o *dataset* ATT48, separando este tempo em três partes, o GPU INIT, tempo necessário para inicialização do OpenCL e carregamento dos dados na GPU pelo JOCL, o TSP, tempo devido ao cálculo do TSP em si e o OUTROS, tempo exigido pelo algoritmo para execução de outros comandos que não envolvem o cálculo em si. Fica evidente que o tempo de partida do GPU INIT tem alto impacto, tomando 42,5% do tempo total, principalmente em um *dataset* tão pequeno, sobrepondo assim os resultados alcançados pelo método.

Figura 9 – Divisão do tempo necessário para processamento do *dataset* ATT48 via GPU (s)



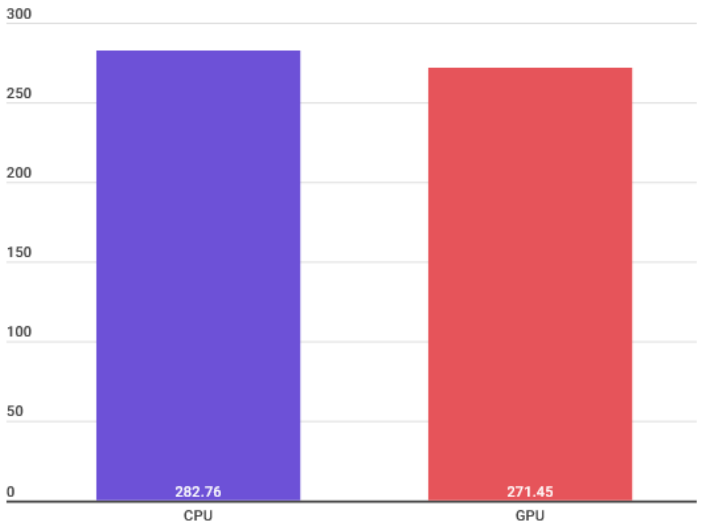
Fonte: JProfiler (2022)

4.1.2 ANÁLISE DOS DEMAIS *datasets*

Os demais *datasets* apresentam redução no tempo de execução tanto no tempo total, quanto no *Self Time*. Em uma análise geral, atribui-se este resultado à melhor utilização dos recursos computacionais, à arquitetura desenvolvida visando este tipo de aplicação e com recursos mais modernos. Abaixo serão apresentados os dados coletados, comparando o tempo total e o *Self Time* para cada cenário analisado.

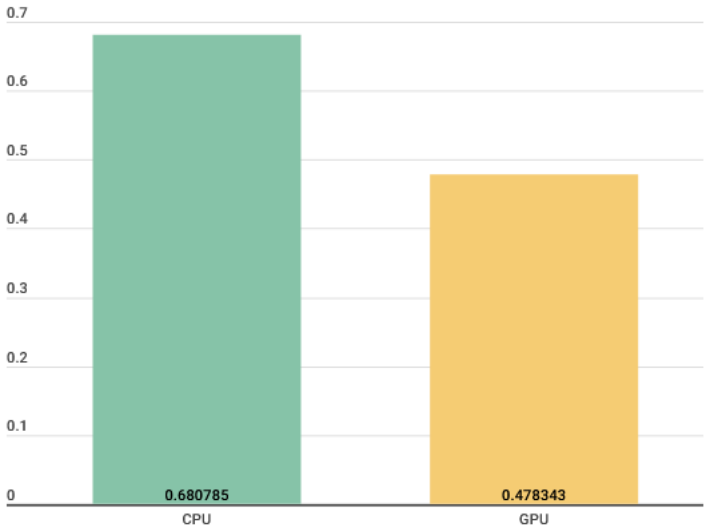
Para o *dataset* RAT575 foi evidenciada uma redução de 11,32 segundos, ou 4,00%, do tempo total de execução conforme demonstrado no gráfico da Figura 10. Analisando o gráfico da Figura 11 identificamos uma redução de 0,202442 ms, ou 27,36%, de redução do tempo necessário para o cálculo da distância Euclidiana.

Figura 10 – Tempos para execução completa do *dataset* RAT575 (s)



Fonte: JProfiler (2022)

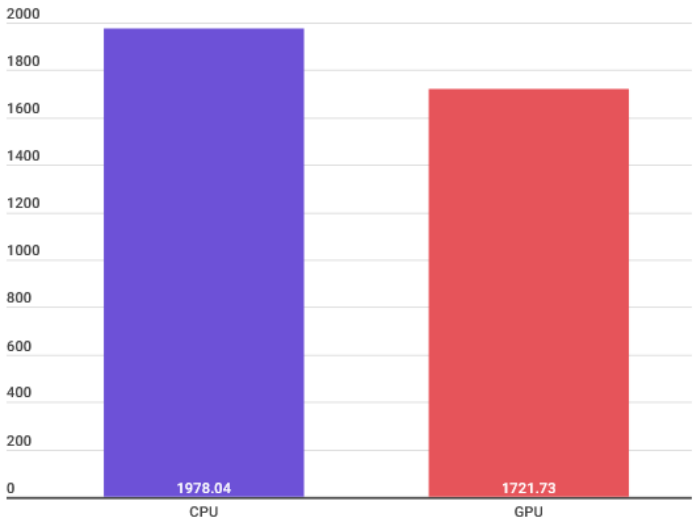
Figura 11 – Tempos para execução da Distância Euclidiana do *dataset* RAT575 (ms)



Fonte: JProfiler (2022)

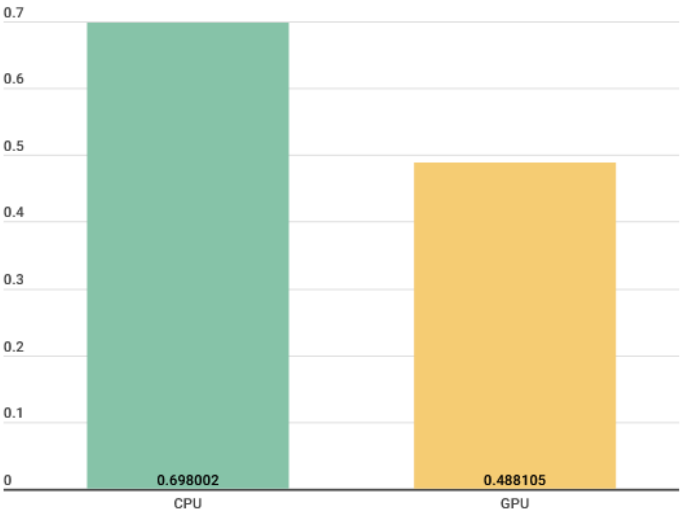
Avançando para o *dataset* NRW1379 foi apontada uma redução de 256,31 segundos, ou 12,96%, do tempo total como indicado no gráfico da Figura 12. Examinando o gráfico da Figura 13 averiguamos uma redução de 0,209897 ms, ou 30,07%, de diminuição do tempo necessário para o cálculo da distância Euclidiana.

Figura 12 – Tempos para execução completa do *dataset* NRW1379 (s)



Fonte: JProfiler (2022)

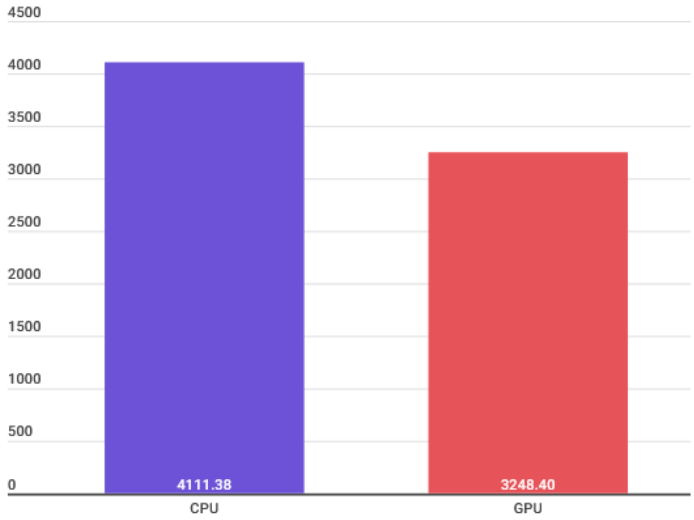
Figura 13 – Tempos para execução da Distância Euclidiana do *dataset* NRW1379 (ms)



Fonte: JProfiler (2022)

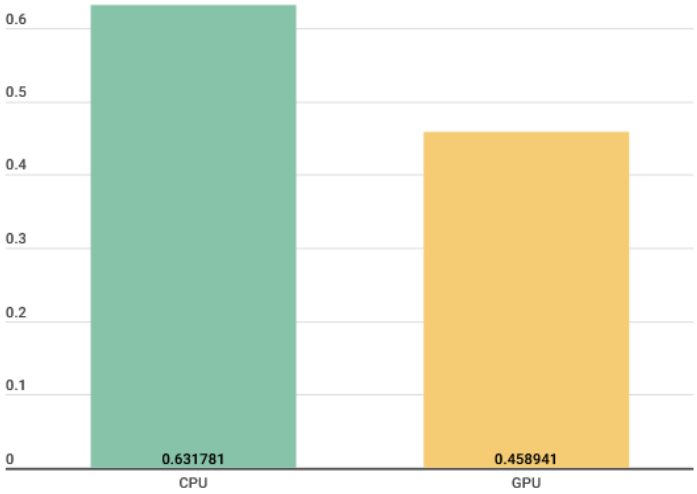
Continuando para o *dataset* BRD14051 foi verificada uma redução de 862,98 segundos, ou 20,99%, do tempo total como exposto no gráfico da Figura 14. Examinando o gráfico da Figura 15 averiguamos uma redução de 0,17284 ms, ou 29,74%, de diminuição do tempo necessário para o cálculo da distância Euclidiana.

Figura 14 – Tempos para execução completa do *dataset* BRD14051 (s)



Fonte: JProfiler (2022)

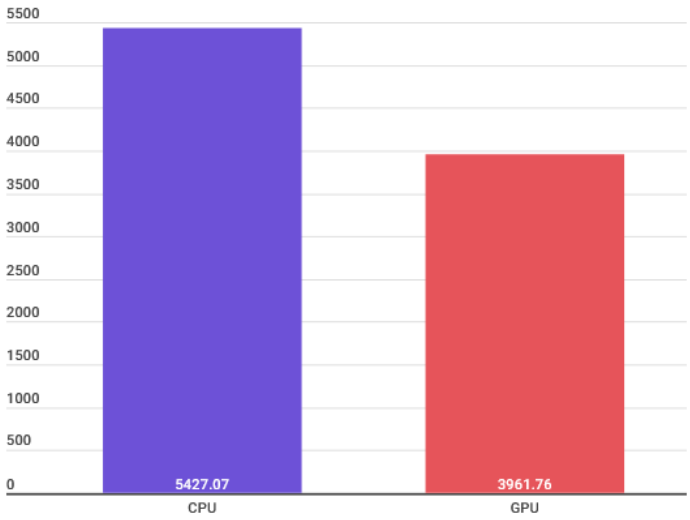
Figura 15 – Tempos para execução da Distância Euclidiana do *dataset* BRD14051 (ms)



Fonte: JProfiler (2022)

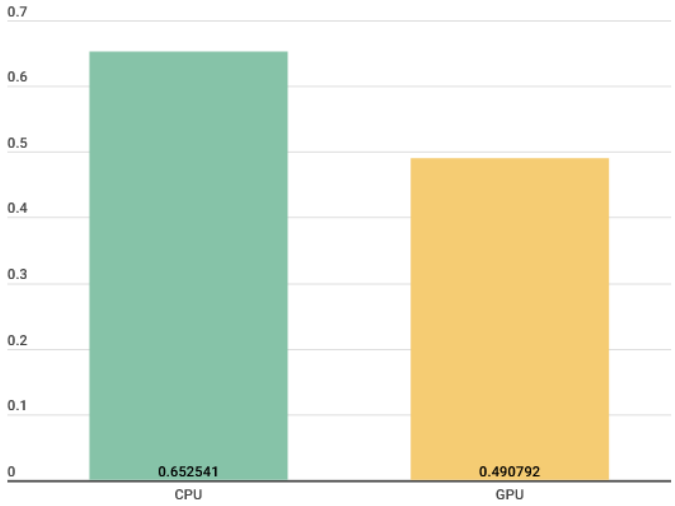
Por fim, o *dataset* D18512 apresentou diminuição de 1465,31 segundos, ou 27,0%, do tempo total como exposto no gráfico da Figura 14. Examinando o gráfico da Figura 15 averiguamos uma redução de 0,17284 ms, ou 30,21%, de redução do tempo necessário para o cálculo da distância Euclidiana.

Figura 16 – Tempos para execução completa do *dataset* D18512 (s)



Fonte: JProfiler (2022)

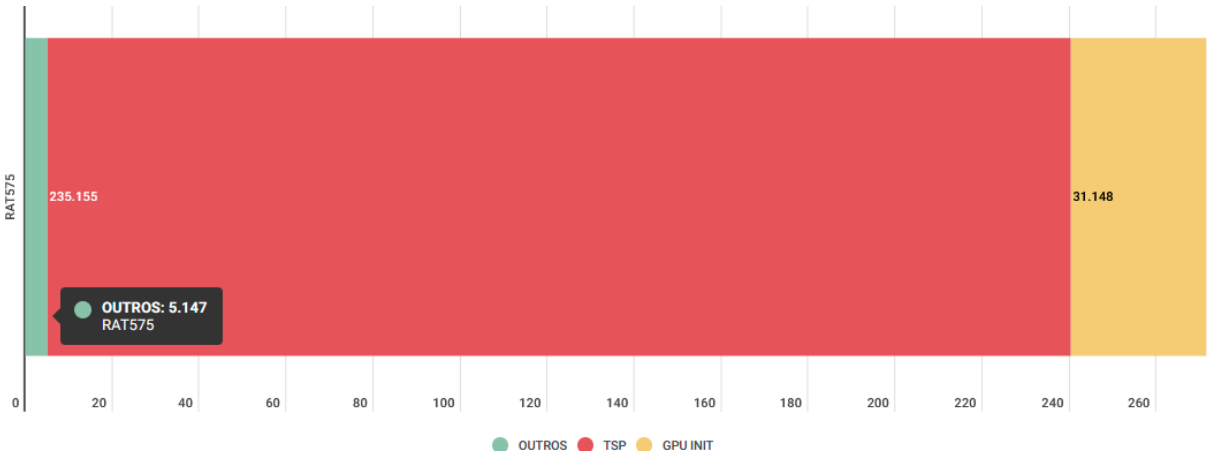
Figura 17 – Tempos para execução da Distância Euclidiana do *dataset* D18512 (ms)



Fonte: JProfiler (2022)

Assim como retratada na Figura 9 para o *dataset* ATT48, foi averiguado o tempo de GPU INIT para os demais *datasets*. A Figura 18 apresenta a divisão dos tempos para o *dataset* RAT575 em que 11,5% do tempo total foi representado pela inicialização da GPU.

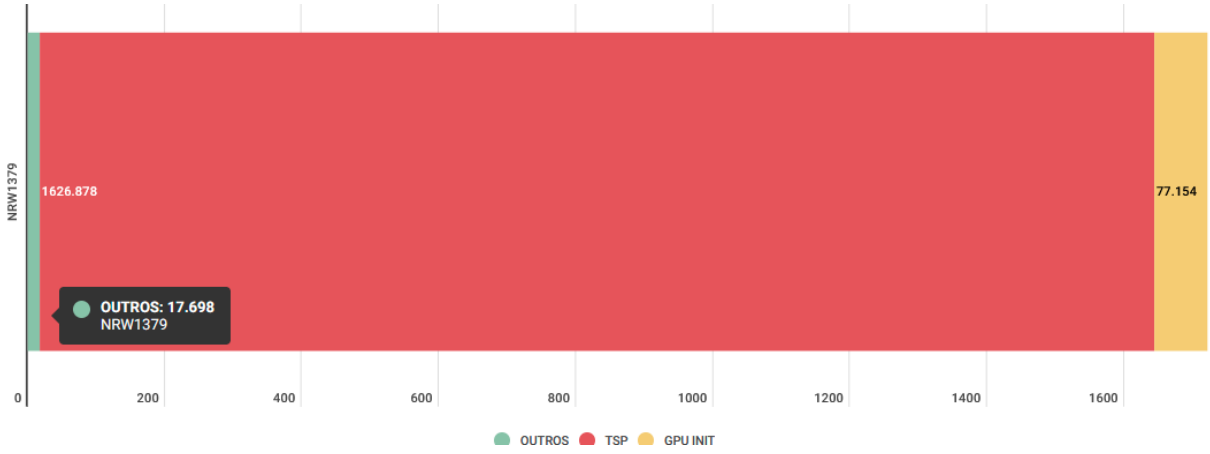
Figura 18 – Divisão do tempo necessário para processamento do *dataset* RAT575 via GPU (s)



Fonte: JProfiler (2022)

A Figura 19 apresenta a divisão dos tempos para o *dataset* NRW1379 em que 4,5% do tempo total foi representado pela inicialização da GPU.

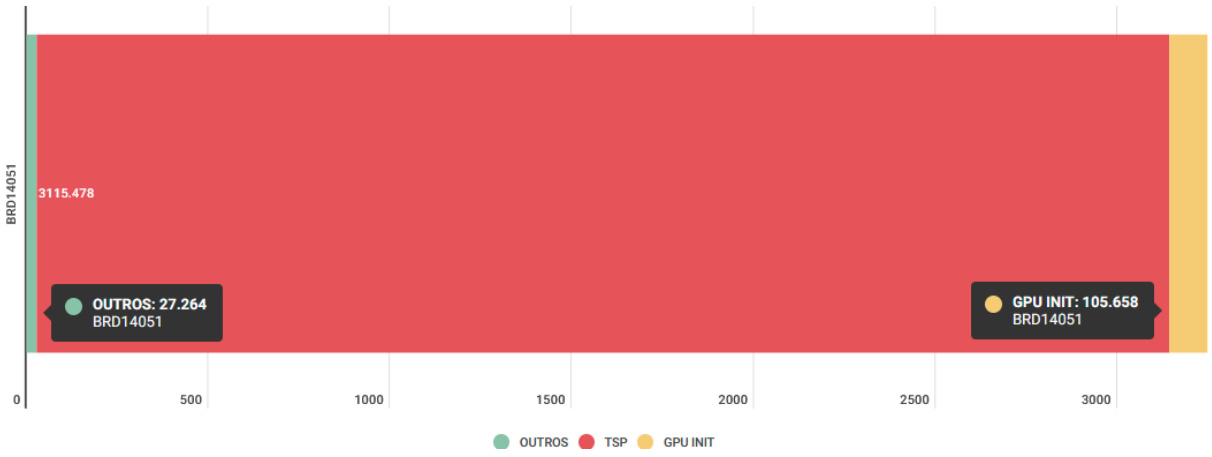
Figura 19 – Divisão do tempo necessário para processamento do *dataset* NRW1379 via GPU (s)



Fonte: JProfiler (2022)

A Figura 20 apresenta a divisão dos tempos para o *dataset* BRD14051 em que 3,3% do tempo total foi representado pela inicialização da GPU.

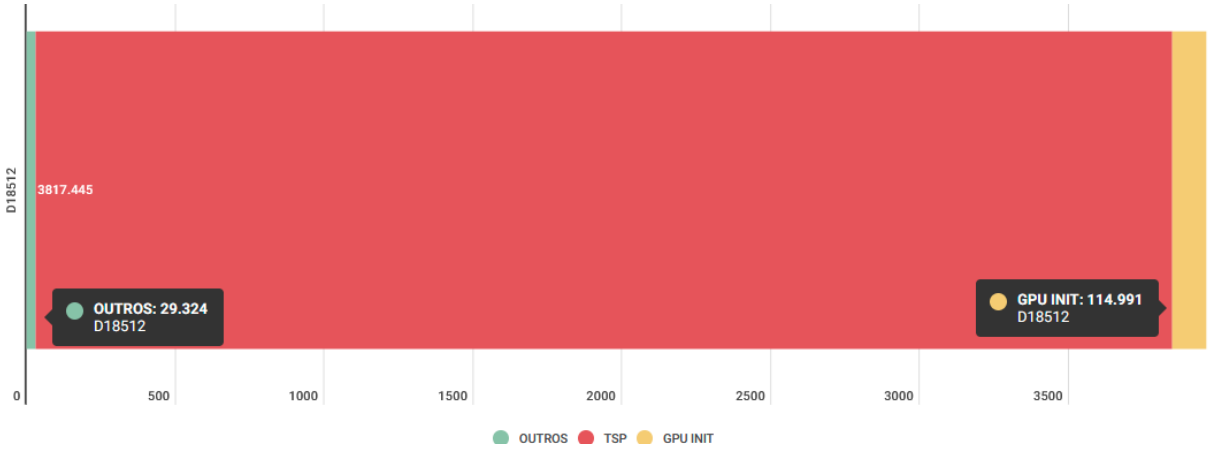
Figura 20 – Divisão do tempo necessário para processamento do *dataset* via GPU (s)



Fonte: JProfiler (2022)

A Figura 21 apresenta a divisão dos tempos para o *dataset* D18512 em que 2,9% do tempo total foi representado pela inicialização da GPU.

Figura 21 – Divisão do tempo necessário para processamento do *dataset* D18512 via GPU (s)



Fonte: JProfiler (2022)

Deste modo, conforme atestado pelos números indicados nos gráficos das Figuras 18, 19, 20 e 21, mesmo que o tempo de GPU INIT se escale conforme o *dataset* cresce, ele se torna ínfimo frente ao tempo total para execução em grandes *datasets* como o D18512, em que este tempo foi de 114,991 segundos, mas representa apenas 2,9% do tempo total, não impedindo que o resultado final seja de um ganho de 27%.

5 CONCLUSÃO

Neste trabalho, foi estudada a aplicação da heurística de Lin-Kernighan para concepção de um algoritmo cujas instruções requisitadas de forma repetitiva e com alto custo de processamento, identificadas através da ferramenta de *profiling* JProfiler, foram paralelizadas com emprego da técnica de *Loop Unrolling* e executadas pela GPU, cuja comunicação fora viabilizada pela biblioteca JOCL. Os resultados experimentais obtidos com a GPU NVIDIA RTX 3090 apresentaram redução considerável no tempo total necessário para o processamento do TSP nos cenários estudados com exceção de uma anomalia encontrada no *ATT48*, cuja razão foi analisada na Subseção 4.1.1. Além do tempo total, foi analisado o *Self Time*, tempo apenas do cálculo da distância Euclidiana, que é a mais repetida e mais custosa do algoritmo, e este por sua vez exibiu ganhos em todos os *datasets* estudados, o que evidencia a efetividade dos métodos propostos nesta produção.

5.1 TRABALHOS FUTUROS

Os estudos realizados nesta pesquisa abrem oportunidades para que trabalhos futuros sejam realizados. Em sua maioria, estes trabalhos são relacionados com estudos mais aprofundados e emprego de outras tecnologias de aceleração de *hardware*. Em primeiro lugar está a comparação dos resultados obtidos neste trabalho com a utilização de FPGA (*Field-Programmable Gate Array*) que é um "circuito integrado que pode ser programado para um uso específico"(TROCHIMIUK, 2022), para executar os cálculos da distância Euclidiana diretamente em circuito lógico a fim de observar possíveis ganhos de desempenho.

A utilização de GPUs da NVIDIA abre também a possibilidade de utilização de CUDA, uma "plataforma de computação paralela para computação de alta performance"(OH, 2012), que apesar de ser uma solução proprietária e só funcionar em dispositivos da NVIDIA, promete ganhos de desempenho ainda maiores em comparação com OpenCL.

Referências

ABDULRAZAQ, M. B. et al. Polynomial reduction of tsp to freely open-loop tsp. In: **2019 2nd International Conference of the IEEE Nigeria Computer Chapter (NigeriaComputConf)**. [S.l.: s.n.], 2019. p. 1–4. Citado na página 3.

AMD. **Processadores para desktop AMD Ryzen™ 9 5950X**. 2022. Disponível em: <https://www.amd.com/pt/products/cpu/amd-ryzen-9-5950x>. Acesso em: 20 de outubro de 2022. Citado na página 11.

BAUMANN, T.; NOACK, M.; STEINKE, T. Performance evaluation and improvements of the pocl open-source opencl implementation on intel cpus. In: **International Workshop on OpenCL**. New York, NY, USA: Association for Computing Machinery, 2021. (IWOCL'21). ISBN 9781450390330. Disponível em: <https://doi.org/10.1145/3456669.3456698>. Citado 2 vezes nas páginas 13 e 15.

BLAZINSKAS, A.; MISEVICIUS, A. Combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem. **Kaunas University of Technology, Department of Multimedia Engineering, Studentu St**, p. 50–401, 2011. Citado 2 vezes nas páginas 4 e 13.

BRäUNL, T. Central processing unit. In: _____. **Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 17–47. ISBN 978-3-540-70534-5. Disponível em: <https://doi.org/10.1007/978-3-540-70534-5_2>. Citado na página 10.

CAIANO, M. M. **Aplicação do Problema do Caixeiro Viajante numa empresa de distribuição: A Heurística de Lin Kernighan**. Tese (Doutorado) — Universidade de Coimbra, 2018. Citado na página 3.

CRIşAN, G. C.; NECHITA, E.; SIMIAN, D. On randomness and structure in euclidean tsp instances: A study with heuristic methods. **IEEE Access**, v. 9, p. 5312–5331, 2021. Citado 2 vezes nas páginas 7 e 12.

CRUCIAL. **DDR Memory Speeds and Compatibility**. 2022. Disponível em: <https://www.crucial.com/support/memory-speeds-compatability>. Acesso em: 25 de outubro de 2022. Citado na página 11.

DANG, J.; ZHANG, Z. A polynomial time evolution algorithm for the traveling salesman problem. In: **2005 International Conference on Neural Networks and Brain**. [S.l.: s.n.], 2005. v. 1, p. 47–49. Citado na página 3.

DAVENDRA, D. **Traveling salesman problem: Theory and applications**. [S.l.]: BoD–Books on Demand, 2010. Citado na página 4.

FISCHER, T.; MERZ, P. A distributed chained lin-kernighan algorithm for tsp problems. In: **19th IEEE International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2005. p. 10 pp.–. Citado na página 3.

FRAHADNIA, F. A new method based on genetic algorithms for solving traveling salesman problem. In: **2009 International Conference on Computational Intelligence, Modelling and Simulation**. [S.l.: s.n.], 2009. p. 11–16. Citado na página 3.

HELSGAUN, K. General k-opt submoves for the lin-kernighan tsp heuristic. **Mathematical Programming Computation**, Physica-Verlag, v. 1, n. 2-3, p. 119–163, 2009. ISSN 1867-2949. Citado na página 5.

HOFFMAN, K. L. et al. Traveling salesman problem. **Encyclopedia of operations research and management science**, Springer New York, v. 1, p. 1573–1578, 2013. Citado na página 3.

INTGRP. Tsp. In: . [s.n.], 2020. Disponível em: <https://github.com/Intgrp/TSP>. Acesso em: 10 de fevereiro de 2022. Citado na página 14.

JPROFILER, E. T. Jprofiler. In: . [s.n.], 2022. Disponível em: <https://www.ej-technologies.com/products/jprofiler/overview.html>. Acesso em: 5 de fevereiro de 2022. Citado 11 vezes nas páginas 11, 12, 17, 18, 19, 20, 21, 22, 23, 24 e 25.

LIN, S.; KERNIGHAN, B. W. **An effective heuristic algorithm for the traveling-salesman problem. Operations Research**. [S.l.]: INFORMS, 1973. Citado na página 3.

MICRON. **GDDR6X: Memory Reimagined**. 2022. Disponível em: <https://www.micron.com/products/ultra-bandwidth-solutions/gddr6x>. Acesso em: 25 de outubro de 2022. Citado na página 11.

NVIDIA. Cuda c++ programming guide. In: . [s.n.], 2021. Disponível em: <https://www.researchgate.net/deref/https%3A%2F%2Fdocs.nvidia.com%2Fcuda%2Fcuda-c-programming-guide%2Findex.html>. Acesso em: 10 de junho de 2022. Citado na página 10.

NVIDIA. **Família GeForce RTX 3090**. 2022. Disponível em: <https://www.nvidia.com/pt-br/geforce/graphics-cards/30-series/rtx-3090-3090ti/>. Acesso em: 20 de outubro de 2022. Citado na página 11.

ODILI, J.; KAHAR, M. N. M.; AHMAD, N. Solving traveling salesman's problem using african buffalo optimization, honey bee mating optimization lin-kerninghan algorithms. **World Applied Sciences Journal**, v. 34, p. 911–916, 01 2016. Citado na página 6.

OH, F. **What Is CUDA?** 2012. Disponível em: <https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/>. Acesso em: 24 de outubro de 2022. Citado na página 26.

OO, N. Z.; CHAIKAN, P. The effect of loop unrolling in energy efficient strassen's algorithm on shared memory architecture. In: **2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)**. [S.l.: s.n.], 2021. p. 1–4. Citado 2 vezes nas páginas 9 e 12.

RAZALI, N. M.; GERAGHTY, J. et al. Genetic algorithm performance with different selection strategies in solving tsp. In: INTERNATIONAL ASSOCIATION OF ENGINEERS HONG KONG, CHINA. **Proceedings of the world congress on engineering**. [S.l.], 2011. v. 2, n. 1, p. 1–6. Citado na página 5.

REINELT, G. Tsplib. In: . [s.n.], 1995. Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/>. Acesso em: 20 de maio de 2022. Citado 2 vezes nas páginas 6 e 7.

REXHEPI, A.; MAXHUNI, A.; DIKA, A. Analysis of the impact of parameters values on the genetic algorithm for tsp. **International Journal of Computer Science Issues**, Volume 10, p. pp 158–164, 01 2013. Citado na página 5.

SAENPHON, T. Enhancing particle swarm optimization using opposite gradient search for travelling salesman problem. **Faculty of Information and Communication Technology, Silpakorn University, Nonthaburi 11120, Thailand**, 9 2018. Citado na página 5.

SANCHES, D. S. **Algoritmos Evolutivos Multi-Objetivo para reconfiguração de Redes de Sistemas de Distribuição de Energia Elétrica**. Tese (Doutorado) — Universidade de São Paulo, 2013. Citado na página 1.

SETHUMADHAVAN, G.; NARAYANASAMY, S. A.; GOPALAKRISHNAN, A. Performance evaluation of image smoothing on cpu and gpu using multithreading — an experimental approach in high performance computing. In: **2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)**. [S.l.: s.n.], 2016. p. 1–5. Citado na página 11.

SINGH, D. R. et al. Genetic Algorithm for Solving Multiple Traveling Salesmen Problem using a New Crossover and Population Generation. **ComputaciÃy Sistemas**, scielomx, v. 22, p. 491 – 503, 06 2018. ISSN 1405-5546. Disponível em: <http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S1405-55462018000200491&nrm=iso>. Citado 2 vezes nas páginas 6 e 12.

SOUSA, L. S. et al. A work proposition method to improve reliability indices. **THE 12th LATIN-AMERICAN CONGRESS ON ELECTRICITY GENERATION AND TRANSMISSION - CLAGTEE 2017**, 2017. Citado na página 1.

SUN, J. et al. Hybrid algorithm based on chemical reaction optimization and lin-kernighan local search for the traveling salesman problem. In: **2011 Seventh International Conference on Natural Computation**. [S.l.: s.n.], 2011. v. 3, p. 1518–1521. Citado na página 3.

SYAMBAS, N. R.; SALSABILA, S.; SURANEGARA, G. M. Fast heuristic algorithm for travelling salesman problem. In: **2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)**. [S.l.: s.n.], 2017. p. 1–5. Citado na página 3.

SYBERFELDT, A.; EKBLOM, T. A comparative evaluation of the gpu vs the cpu for parallelization of evolutionary algorithms through multiple independent runs. **International Journal of Computer Science and Information Technology**, v. 9, p. 01–14, 06 2017. Citado 2 vezes nas páginas 10 e 13.

TROCHIMIUK, M. **FPGA programming – how it works and where it can be used**. 2022. Disponível em: <https://codilime.com/blog/fpga-programming-how-it-works-and-where-it-can-be-used/>. Acesso em: 23 de outubro de 2022. Citado na página 26.

TSUDA, F. **Utilização de técnicas de GPGPU em sistema de vídeo-avatar**. Tese (Doutorado) — Universidade de São Paulo, 2011. Citado na página 10.

WEI, X. et al. Multi-core-, multi-thread-based optimization algorithm for large-scale traveling salesman problem. **Alexandria Engineering Journal**, v. 60, n. 1, p. 189–197, 2021. ISSN 1110-0168. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1110016820303227>. Citado 2 vezes nas páginas 8 e 10.

ZAPARANUKS, M. H. D. Algorithmic profiling. **University of Lugano**, 2018. Citado na página 11.